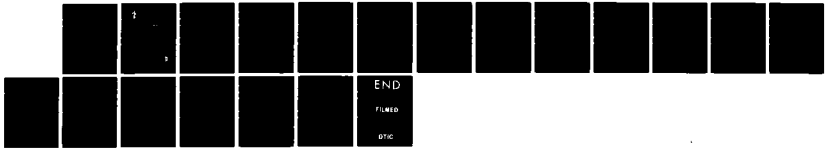
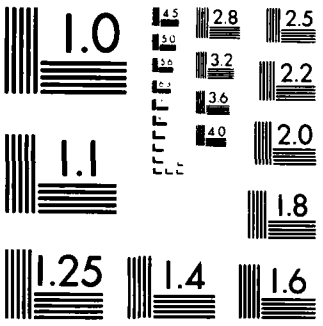
RD-A149 441 CHECKING MICROCODE ALGEBRAICALLY(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) J M FOSTER OCT 84 RSRE-MEMO-3748 DRIC-BR-93876 1/1

UNCLASSIFIED F/G 12/1 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNLIMITED

RSRE
MEMORANDUM No. 3748

ROYAL SIGNALS & RADAR ESTABLISHMENT

CHECKING MICROCODE ALGEBRAICALLY

Author: J M Foster

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

RSRE MEMORANDUM No. 3748

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3748

Title: CHECKING MICROCODE ALGEBRAICALLY

Author: J M Foster

Date: October 1984

SUMMARY

This Memorandum describes algebraic methods used by a program which processes microcode based on the AMD 2910 in order to find some of its properties. The methods could be applied to other controllers. Examples of the properties which can be found are given, including checking for timing constraints, ensuring that interrupts are polled frequently, checking against expression stack overflow and ensuring the absence of certain sequences of instruction. The method separates into a part which deals only with the control structure, for this program that of the AMD 2910, and a part which deals with the operations performed by the micro-instructions, in this case those of the ICL Perq. It has been used to check many properties of the implementation of Flex on Perq, which involves more than 4000 microinstructions.

RSRE MEMORANDUM 3748

CHECKING MICROCODE ALGEBRAICALLY

J M Foster

LIST OF CONTENTS

1 INTRODUCTION

Many authors have described the use of algebraic methods to discover properties of programs (1),(2),(3),(4),(6),(7),(9),(10) and (11). These methods have been most often used for optimisation in compilers but it is also possible to use them to prove the absence of wide classes of error in programs, or if these errors are present it may be possible to locate them.

Microcode is a particularly suitable object for such studies. The correctness of the whole computer depends on the correctness of the microcode, so the expense of guaranteeing the absence of certain errors can easily be borne. Microcode is often written in a very dense and compact fashion, with many GOTOs and much sharing of code because of the overriding need for speed and space economy and it is also often subject to unusual constraints of timing or impermissible instruction sequences. It is therefore particularly prone to errors which are difficult to find by visual inspection but can be found by algebraic methods. Furthermore, microcode usually has a fairly simple control structure, conditional jumps, some kinds of switch jumps including one for instruction entry, subroutine calls, returns and exceptions account for most microcode control structure. Procedures with parameters are usually absent and there are a fixed number of registers, main store being regarded as a peripheral device. All this makes the algebraic method easy to apply though subroutines and exception handling do need an extension of the usual technique.

The algebraic methods take account of the control structure of the program, without usually being able to consider the particular values that are being manipulated. It is not possible for example, to know which way a conditional jump will go, we merely know that it will go one way or the other. Hence the methods are pessimistic. If they say that an optimisation is possible, then that is certainly so, but the optimisation might be possible without being detectable, since the actual values which can possibly occur might rule out some paths which the method cannot afford to ignore. Likewise, if the program is said to be free from a certain sort of error, then that is so, but the presence of an error might be suggested when in fact it could be ruled out by a more detailed examination of the semantics.

The method factors into two parts. In one, which is independent of the particular property being investigated, we calculate an algebraic expression corresponding to the control structure of the microcode program. In the other we calculate a function of that algebraic expression which gives us the property we require. So the control structure of the microcode is entirely processed in the first component and is divorced from the particular property under investigation which belongs entirely to the second component.

This paper describes the method, shows how it may be used to find various kinds of error in microcode and gives an account of a program which implements the method for microcode based on the AMD 2910 microcontroller. It could be adapted for other controllers.

We start by giving an outline of the method in the rest of this section. The second describes the program for the AMD 2910 microcontroller. In the third section are examples of the method applied to finding a number of apparently quite different and useful properties of microcode. These include ensuring that an expression stack cannot overflow, finding the maximum time between polling for interrupts, checking that timing constraints on the use of store are met and making sure that the interrupt routine does not disturb values in machine registers. The fourth section contains some conclusions. An appendix gives the formal definition or regular algebras.

We use what is known as regular algebra. We can think of this as being defined abstractly, by giving the operations of the algebra and saying what laws they must obey. Such a definition is given in the appendix. We can also think of various concrete realisations of the algebra. For one such realisation the values manipulated by the algebra are sets of paths starting at one point of a program and ending at another. The operations are ways of combining these to make larger sets of paths and ultimately the whole program.

An elementary regular expression, one instruction of the microcode, corresponds to a path of one step. One operation of the algebra, denoted by "." and read "followed by", takes two regular expressions and makes a new one consisting of all the paths going through the first parameter of the operation and then through the second. A second operation, denoted by "+", takes two regular expressions and makes a new one consisting of all the paths through either one or the other. There is a special regular expression, written "1", which corresponds to a path of zero steps, in other words a short circuit between the starting and finishing points. A third and final operator, "*", takes a regular expression, r, and produces the regular expression

$$1 + r + r.r + r.r.r + \ldots$$

that is, the paths in the result are those obtained by going zero times round r, or once, or twice etc. We may sometimes also use a special value, written "0" which corresponds to no path, but this is not a necessary part of the definition, just one which is sometimes useful.

There are various laws which are obeyed by this algebra of sets of paths. For example

$$a.(b+c) = a.b + a.c$$

which is a distributive law and

$$a.(b.c) = (a.b).c$$

which is the associative law for "followed by". These are examples of the laws necessary for regular algebras.

It is possible to take microcode and produce from its control structure a regular expression which describes the possible paths through it. While this is being done various checks can be made on the structure of the microcode. For example, a path from the start which must loop for ever or a disconnected piece of microcode can be detected and reported. In such cases a regular expression is not produced and the programmer is sent back to correct his microcode. The interpretation of the microcode can be done independently of its later use for checking and we therefore need just one program to do it.

As well as the realisation of the operations and laws of regular algebras in terms of paths, we could realise them in terms of other values and operations. Such a realisation would again define values to be manipulated, operations +', .' and *' and a unit, 1', all of which obey the laws. A homomorphism of regular algebras, H, is a mapping which obeys the rules

$$H(1) = 1'$$

$$H(a.b) = H(a) .' H(b)$$

$$H(a+b) = H(a) +' H(b)$$

$$H(a*) = H(a) *'$$

In order to obtain a property of the program we will define a homomorphism of regular algebras from the path algebra to an algebra devised especially for determining the particular property. The implication of the rules for homomorphisms above is that we can calculate the property of a composite program from the properties of its parts. It may require some ingenuity to think of a suitable homomorphism, but it is a skill which comes with practice and the rewards are considerable.

A property of a program will be some sort of value. It may be as simple as a truth value, saying that the program is good or bad, or an integer giving the maximum value of something, or it may be quite a complex structure. The path algebra is such that if we choose how the one-step values are mapped, that is if we say what is the property of the individual instructions, then there is only one way of extending this to a homomorphism. If we introduce

a function, atom, which takes the one-step values to their properties, then the rule

H(a) = atom(a), when a is a single instruction

together with the rules given above, define a recursive program for evaluating the property and if this is applied to the regular expression derived from the microcode it will tell us the property of the microcode.

Let us take a very simple example to illustrate this. Suppose that some of the elementary instructions can be wrong. We wish to find out whether a program contains a wrong instruction on any of its paths. We take the type of our property values to be truth values, true being interpreted as meaning that a set of paths has a wrong instruction in it and false as meaning that it contains no wrong instruction. A short circuit has no error so 1' is false. The function atom is that which says for each elementary instruction whether it is good or bad. For .' we choose "or" since we want to detect an error if there is one in either component and for +' we also choose "or" since we want to detect an error if there is one in either sequent. The operation *' can be deduced to be the identity operation on truth values. We must check that these operations obey the laws for regular algebras which are set out in the appendix. Then we can obtain the truth value which tells us whether there was an error by evaluating the homomorphism for the regular expression produced from the microcode.

2 EVALUATION OF THE HOMOMORPHISMS

2.1 STRUCTURE OF THE PROGRAM

A program has been written in ALGOL 68 to implement the method for microcode written for the ICL Perq computer. The Perq microcode is based on the AMD 2910 microcontroller and this provides the control structure of the microcode, with a few special features added which are peculiar to the Perq. Since the production of the regular expression depends only on the control structure, this part of the program depends on the properties of the AMD 2910, though the individual homomorphisms will depend on the other parts of the microinstructions. So the major part of the program is dependent on the AMD 2910 rather than the whole of Perq. Methods used by other authors are described in (2) and (10).

The program first assembles the text of the microcode into an in-store representation. This, of course, depends on the form of written microcode for Perq. It then applies a given homomorphism

hom(assemble(text), unit, dot, plus, star, atom)

the result of which is the value of the homomorphism for the whole program. The value, unit, and the functions, dot, plus, star and atom define the particular homomorphism which is to be used and the procedure, hom, knows the control structure of the machine and evaluates the regular expression and its homomorphism. So assemble, unit, dot, plus, star and atom understand about the operations of the Perq microcode, but hom only knows about the control structure provided by the AMD 2910. It would have been possible to evaluate the regular expression first, and then the homomorphism afterwards, but that would have meant keeping a representation

of the regular expression which for a large microcode would have been very bulky. The program therefore evaluates the homomorphism as it goes, since this usually takes very much less space.

Since different homomorphisms will map to regular algebras in terms of different base types (modes in ALGOL 68), the types of unit, dot, plus, star, atom and hom will be different when calculating different properties. ALGOL 68 does not permit modes to be parameterised so it is necessary to recompile in order to generate a program which applies a new homomorphism. The structure is, schematically:

```
BEGIN

    Mode M = definition of mode of values produced by hom

        PROC atoms = (INSTRUCTION i)M:  body;
        PROC dot   = (M a, b)M:  body;
        PROC plus  = (M a, b)M:  body;
        PROC star  = (M a)M:  body;
        M unit = value;

    PROC hom = (MICROCODE m,
                M u,
                PROC(M, M)M d,
                PROC(M, M)M p,
                PROC(M)M s,
                PROC(INSTRUCTION)M atom)
               M:

                body;

                hom(assemble(text), unit, dot, plus, star, atom)

END
```

The first six lines of definition are special to a particular homomorphism, the remaining lines are always the same. Hom only uses the control part of the microcode.

Of course it is possible for the control structure of the microcode to be illegal itself. For example, the AMD 2910 chip has only five levels of subroutine entry available, so exceeding this number must be wrong, excluding any consideration of regular expressions. Also if we start to interpret a new macroinstruction when the subroutine stack is not empty this is likely to be a mistake. These and other errors are detected by hom while it is evaluating the regular expression and are indicated as errors to the user.

2.2 FROM CONTROL STRUCTURE TO REGULAR EXPRESSIONS

From the point of view of rapid calculation, the most important properties of regular algebras are the associative and distributive laws. These imply that we can calculate the regular expression corresponding to a part of the program and slot it into place in a larger expression, without any problems about the order in which these things are done. So we can calculate the regular expression starting from a

label in the code up to, say, the points at which we start to decode the next instruction, and store it, or rather its homomorphic image, together with the label. Then when we find another jump to that label we can use the already computed value. Something similar can be done for subroutine calls by storing the value associated with the subroutine up to the returns and exception jumps, and re-using this. This is made complex by the need to treat exceptions (Jump-pop in the AMD 2910) and because of the explicit manipulations of the subroutine link stack which are possible. Subroutine calls, exceptions and the efficiency of the calculation of regular expressions are dealt with in another paper, (5).

Clearly a section of program which continues to another using "next" or an unconditional jump is to be composed using the "followed by" operator.

A conditional jump brings the + operator into play. Notice that the operation performed by the instruction before jumping has to be dealt with, so a conditional jump yields

(effect of instruction) . (dest1 + dest2)

where dest1 and dest2 are the regular expressions calculated from the continuation and the (labelled) destination. Switch jumps are dealt with similarly.

Microcode typically has a structure

initiate.(ins0 + ins1 + ins2 ...)*

where initiate is microcode to set up the system, and ins0, ins1 etc are microcode to interpret the various macroinstructions. This, therefore, is how we treat the switch jump to decode the next instruction.

The AMD 2910 has a register S. This is used both for counting, which can be treated by the same method as conditional jump, and also to hold a destination for a jump. This opens the possibility of computed jumps, but if the values which are loaded into S are constants of the microcode it is possible to calculate the values which can be in S at any moment and treat a jump using S as if it were a switch jump to all these places.

We have still to deal with loops caused by jumps. Let us take a note when we start to process code starting at a label. If, while processing it, we arrive at a jump to the label again we have in effect the situation

x = a + b.x

where a is the rest of the regular expression whether complete or not, and b is the set of paths leading up to the return to the label. For this we produce the regular expression

b* . a

It can be shown that this substitution can always be made (see Appendix), and that although doing things in a different order may lead to different regular expressions, these are always equivalent under the laws of regular algebras. It is at this point that loops from which there is no escape can be detected.

The rest of the features of the AMD 2910 can be dealt with in ways which are easily deduced from those which have been treated above.

3 EXAMPLES OF HOMOMORPHISMS

3.1 SOME REGULAR ALGEBRAS

We start by considering a few examples of regular algebras in order to help with the homomorphisms to follow.

Consider the integers with

$$a.b \to a+b \qquad \text{(addition on integers)} \qquad (3.1)$$

$$a+b \to \max(a,b)$$

$$a^* \to \text{if } a > 0 \text{ then } \infty \text{ else } 0$$

$$1 \to 0$$

$$0 \to -\infty$$

This will satisfy the laws for regular algebras with a zero (see Appendix) provided that we take $-\infty + \infty = -\infty$.

Dually we can take

$$a.b \to a+b \qquad \text{(addition on integers)} \qquad (3.2)$$

$$a+b \to \min(a,b)$$

$$a^* \to \text{if } a < 0 \text{ then } -\infty \text{ else } 0$$

$$1 \to 0$$

$$0 \to \infty$$

This will satisfy the laws provided that we take $-\infty + \infty = \infty$.

Another simple example is to take truth values, T and F with

$$a.b \to a \text{ and } b \qquad (3.3)$$

$$a+b \to a \text{ and } b$$

$$a^* \to T$$

$$1 \to T$$

$$0 \to F$$

or its obvious dual.

Several of the examples of homomorphisms below use regular algebras derived in the following way. Given a regular algebra with a zero defined on a set R, we can obtain a regular algebra defined on R x R by

$$(a1,b1).(a2,b2) \rightarrow (a2+a1.b2,\ b1.b2) \qquad (3.4)$$

$$(a1,b1) + (a2,b2) \rightarrow (a1+a2,\ b1+b2)$$

$$(a,b)^* \rightarrow (a.b^*, b^*)$$

$$1 \rightarrow (0,1)$$

We can see that (0,0) is a right zero for "followed by", but it is not a left zero. If a zero is needed one may be added consistently to the base set, but it will not be needed in any example below.

3.2 EXAMPLES OF HOMOMORPHISMS

Consider the following simple example, useful in what follows. We wish to find whether a path expression contains a path of zero length, that is, whether there is a possible short-circuit between start and finish. Note that the path expressions that we produce from microcode are slightly different from the sequence of instruction steps through the program. For the single instruction

label : op, goto label

translates into the path expression

op. op*

since the operation is always obeyed at least once. So a zero length path in a path expression is quite possible. Let us take R to be truth values, with

$$\begin{aligned} \text{atom}(a) &= \text{false} \qquad (3.5)\\ 1 &= \text{true}\\ a.b &= a \text{ and } b\\ a+b &= a \text{ or } b\\ a^* &= \text{true}\\ 0 &= \text{false} \end{aligned}$$

It can easily be seen that this computes a correct answer and satisfies the laws for regular algebras with a zero. It just uses the algebra of formula 3.3.

Now let us design a homomorphism which will decide whether an X instruction is immediately followed by a Y instruction. Thought it appears that this might be easy to detect by eye, in fact to do so in 4000 instructions with all the possibilities of interaction which are permitted by the AMD 2910, and to be certain that no example has been missed, is no light task. It is something which is useful for Perq microcode, since there are instructions which can occasionally, because of interrupts, spoil the effect of the following one. We will let R consist of quadruples of truth values, (x, y, t, e), in which x will tell us if any of the paths ends with an X instruction, y will say whether any path starts with a Y instruction, t will say whether there is a path of zero length, and e will tell us whether there is an example of XY in any of the paths, which is the error we are looking for. Let

```
atom(X)  =  (T, F, F, F)                                   (3.6)
atom(Y)  =  (F, T, F, F)
atom(a)  =  (F, F, F, F) otherwise
      1  =  (F, F, T, F)

(x1, y1, t1, e1) . (x2, y2, t2, e2) =

                (x2 or (t2 and x1) ,
                 y1 or (t1 and y2) ,
                 t1 and t2 ,
                 e1 or e2 or (y1 and x2))

(x1, y1, t1, e1) + (x2, y2, t2, e2) =

                (x1 or x2 ,
                 y1 or y2 ,
                 t1 or t2 ,
                 e1 or e2)

(x, y, t, e)* = (x, y, T, e or (x and y))
```

The t component is just the same as in formula (3.5). The pair x and t as well as the pair y and t are examples of formula (3.4). The laws for e can also be easily checked. Inspection will verify that these definitions are a correct interpretation of what was required. Note that it might be the case that an instruction X contains a conditional jump which, because of extra facts which we know, but the regular algebra does not know, cannot jump to the Y instruction. In this case we shall be told that there is an error when there is none. But certainly if this homomorphism says there is no error, we can be sure that this is so. In practice we would like, not only to know that the program is free from this error if it is so, but also if there is an error we would like to be told where the fault occurred. Such error location can be done in this case without much difficulty by adding extra components to R, but is has not been done here, in order to keep the example uncluttered. However, in general, it is not so easy because the error is not usually easily attributed to a particular place in the program.

Another example, closely related to formula (3.5), will give us the shortest path through a program in terms of the number of instructions. This could easily be modified to give the minimum time. Take R to be integers and let

$$
\begin{aligned}
\text{atom}(a) &= 1 \\
1 &= 0 \\
a.b &= a+b \quad \text{(integer addition)} \\
a+b &= \min(a,b) \\
a^* &= 0 \\
0 &= \infty
\end{aligned}
\tag{3.7}
$$

This is just the regular algebra of formula (3.2), remembering that the number of steps cannot be negative. From this we can derive a homomorphism which will check for the following kind of error. It happens on the Perq that if a certain sort of store instruction is followed within four instructions by another memory instruction then the memory instruction will go wrong. Ensuring that this does not happen is clearly both important and difficult to check. Let us devise a homomorphism to ensure that an instruction of class X is not followed within n steps by an instruction of class Y. We take R to consist of three integers and a truth value and interpret (x, y, t, e) by letting x be the smallest number of steps from an X to the end, Y the smallest number of steps from the start to a Y, t the shortest number of steps through the path expression not involving X or Y, and e the truth value telling whether an error has occurred. Let

$$
\begin{aligned}
\text{atom}(X) &= (0, \infty, \infty, F) \\
\text{atom}(Y) &= (\infty, 0, 1, F) \\
\text{atom}(a) &= (\infty, \infty, 1, F) \quad \text{otherwise}
\end{aligned}
\tag{3.8}
$$

$$
\begin{aligned}
&(x1, y1, t1, e1) \,.\, (x2, y2, t2, e2) = \\
&\qquad (\min(x2,\ x1+t2)\ , \\
&\qquad\ \min(y1,\ t1+y2)\ , \\
&\qquad\ t1+t2\ , \\
&\qquad\ e1 \text{ or } e2 \text{ or } x1+y2 < n+1)
\end{aligned}
$$

$$
\begin{aligned}
&(x1, y1, t1, e1) + (x2, y2, t2, e2) = \\
&\qquad (\min(x1,x2)\ , \\
&\qquad\ \min(y1,y2)\ , \\
&\qquad\ \min(t1,t2)\ , \\
&\qquad\ e1 \text{ or } e2)
\end{aligned}
$$

$$
(x, y, t, e)^* = (x, y, 0, e \text{ or } x+y < 5)
$$

This is clearly closely related to formula (3.6) and again is an example of the use of formula (3.4).

We may find the maximum number of steps (or time) through a program by using the algebra of formula (3.1) and taking

$$
\text{atom}(a) = 1
$$

We can use this in a similar way to find the maximum distance between instances of a particular kind of instruction, X. This is useful, for example if we want to ensure that polling for interrupts occurs sufficiently frequently to keep up with some peripheral. We take a quadruple of integers, (a, b, t, m), a being the maximum distance after an X, b the maximum distance before, t the maximum distance through the path expression not involving an X, and m the maximum separation between Xs

$$
\begin{aligned}
\text{atom}(X) &= (0,\ 0,\ -\infty,\ -\infty) \\
\text{atom}(a) &= (-\infty,\ -\infty,\ 1,\ -\infty) \\
1 &= (-\infty,\ -\infty,\ 0,\ -\infty)
\end{aligned}
\tag{3.9}
$$

$$
\begin{aligned}
(a1, b1, t1, m1) \cdot (a2, b2, t2, m2) = \\
&(\max(a2,\ a1+t2)\ , \\
&\ \max(b1,\ b2+t1)\ , \\
&\ t1+t2\ , \\
&\ \max(m1,\ m2,\ a1+b2))
\end{aligned}
$$

$$
\begin{aligned}
(a1, b1, t1, m1) + (a2, b2, t2, m2) = \\
&(\max(a1,\ a2)\ , \\
&\ \max(b1,\ b2)\ , \\
&\ \max(t1,\ t2)\ , \\
&\ \max(m1,\ m2))
\end{aligned}
$$

$$
\begin{aligned}
(a, b, t, m)^* = \ &(\text{if } t > 0 \text{ then } a + \infty \text{ else } a\ , \\
&\ \text{if } t > 0 \text{ then } b + \infty \text{ else } b\ , \\
&\ \text{if } t > 0 \text{ then } \infty \text{ else } 0\ , \\
&\ \text{if } t > 0 \text{ then } \max(m,\ a+b+\infty) \text{ else } \max(m,\ a+b))
\end{aligned}
$$

Since any loop is likely to give a value of infinity this will mean that we must look at every loop which does not poll for interrupts in order to check by hand that using it will not exceed the permitted interval. However, the homomorphism can be modified to find such loops, and it is in any case probably wise to look at them.

We will produce a homomorphism to check that any sequence of instructions starting with an X instruction and ending with a Z instruction does not contain any Y instructions. This might be because Y spoils something set up by X for Z to use. Typically Y is the interrupt poll. We use a septet of truth values

$$(x,\ y,\ z,\ xy,\ yz,\ t,\ e)\ ,$$

Let x mean that there is a path from an X to the end of the path expression not involving a Y or a Z, let y mean that there is a path from start to finish with one or more Y instructions on it but neither an X nor a Z, and let z mean that there is a path from the start to a Z instruction without X or Y. The truth value xy shall mean that there is a path with an X on it followed by some Ys and dually for yz. Let t mean that there is a path without X, Y or Z and e shall signify that an error has been detected.

```
atom(X)  =  (T, F, F, F, F, F, F)                    (3.10)
atom(Y)  =  (F, T, F, F, F, F, F)
atom(Z)  =  (F, F, T, F, F, F, F)
atom(a)  =  (F, F, F, F, F, F, F)  otherwise
     1   =  (F, F, F, F, F, T, F)

(x1, y1, z1, xy1, yz1, t1, e1) . (x2, y2, z2, xy2, yz2, t2, e2)  =

     (x2 or (x1 and t2) ,
     (y1 and t2) or (t1 and y2) or (y1 and y2),
      z1 or (z2 and t1) ,
      xy2 or (xy1 and t2) or (xy1 and y2) or (x1 and y2) ,
      yz1 or (t1 and yz2) or (y1 and yz2) or (y1 and z2) ,
      t1 and t2 ,
      e1 or e2 or (x1 and yz2) or (xy1 and z2))

(x1, y1, z1, xy1, yz1, t1, e1) + (x2, y2, z2, xy2, yz2, t2, e2)  =

     (x1 or x2 ,
      y1 or y2 ,
      z1 or z2 ,
      xy1 or xy2 ,
      yz1 or yz2 ,
      t1 or t2 ,
      e1 or e2)

(x, y, z, xy, t, e)*  =  (x ,
                          y ,
                          z ,
                          xy or (x and y) ,
                          yz or (y and z) ,
                          T ,
                          e or (x and yz) or (xy and z) or
                         (x and y and z))
```

Finally, we show how we may ensure that the limits of an expression stack are not exceeded. Suppose we have two instructions, push and pop, and a stack with is limited to n items. The instructions only change the number of items in a stack, so we work in terms of change relative to the start, and will assume that the stack is initialised at the start of the program. Take a quartet of integers (d, i, s, g), d being the greatest decrease in stack size from start to finish, i being the greatest increase, s being the smallest number of items in the stack relative to the start which occurred anywhere in the paths, and g is the greatest.

```
atom(push)  =  (1, 1, 1, 1)                          (3.11)
atom(pop)   =  (-1, -1, -1, -1)
  atom(a)   =  (0, 0, 0, 0)  otherwise
       1    =  (0, 0, 0, 0)

(d1, i1, s1, g1) . (d2, i2, s2, g2)  =

                  (d1+d2 ,
                   i1+i2 ,
                   min(s1, s2+d1) ,
                   max(g1, g2+i1)
```

$$(d1, i1, s1, g1) + (d2, i2, s2, g2) =$$

$$(\min(d1, d2),\ \max(i1, i2),\ \min(s1, s2),\ \max(g1, g2))$$

$$(d, i, s, g)^* = (\text{if } d<0 \text{ then } -\infty \text{ else } 0,\ \text{if } i>0 \text{ then } \infty \text{ else } 0,\ \text{if } d<0 \text{ then } -\infty \text{ else } s,\ \text{if } i>0 \text{ then } \infty \text{ else } g)$$

If the final value of g is greater than n then there is an error, and if the final value of s is less than 0 there is an error. Pinpointing the error can be more difficult since it is not necessarily a localised mistake, but the area where the maximum or minimum occurred can be found. It is more complex to try to find the same kind of mistake in the presence of a stack-reset operation, because the natural extension of formula 3.11 does not obey the distributive law. However, a modification of this method can be designed which does work.

4 CONCLUSIONS

We have given a number of examples of useful diagnostic properties of microcode that can be computed using the method of homomorphisms and show how a porogram has been written which separates this calculation between parts involving the control structure and thus depending only on the AMD 2910 and parts involving knowledge of the Perq microcode. This program runs fairly quickly and has proved its use in removing errors from about 4000 instructions of Perq microcode implementing the Flex architecture. Most of the tests which were run found errors, many of which were subtle, involving interrupts or unlikely combinations of circumstance which would have been difficult to remove by the usual methods of trial and error.

ACKNOWLEDGEMENTS

I would like to acknowledge many helpful remarks from B D Bramson on an earlier draft of this paper.

REFERENCES

1 Aho A V, "Principles of compiler design", Addison-Wesley, 1977.

2 Backhouse R C and Carre B A, "Regular algebra applied to path-finding problems", J Inst Math Applic Vol 15, pp161-186, 1975.

3 Bramson B D and Goodenough S J, "Data use analysis for computer programs", unpublished RSRE report.

4 Bramson B D, "Information flow analysis for computer programs", unpublished RSRE report.

5 Foster J M, "Regular expression analysis of procedures and exceptions", RSRE Memo No 3749.

6 Kam J B and Ullman J D, "Monotone data flow analysis frameworks", Acta Inf Vol 7, pp305-317, 1977.

7 Rosen B K, "Monoids for rapid data flow analysis", SIAM J Comput Vol 9, pp159-196, 1980.

8 Salomaa A, "Two complete axiom systems for the algebra of regular events", Journal ACM Vol 13, No 1, pp158-169, Jan 1966.

9 Schaeffer M, "A mathematical theory of global program optimisation", Prentice-Hall 1973.

10 Tarjan R E, "A unified approach to path programs", J ACM, Vol 28, No 3, pp577-593, July 1981.

11 Wegbreit B, "Property extraction in well-founded property sets", IEEE Trans Software Eng Vol 1, pp270-285, 1975.

APPENDIX 1

The definition of regular algebras can be expressed formally[8]. A regular algebra, R, consists of an alphabet, A, of atomic values, a unit value, 1, and three operations ".", "+", and "*".

$$. : R \times R \rightarrow R$$

$$+ : R \times R \rightarrow R$$

$$* : R \rightarrow R$$

The following laws are satisfied:

(a.b).c	=	a.(b.c)	associativity of .
(a+b)+c	=	a+(b+c)	associativity of +
a+b	=	b+a	commutativity of +
a+a	=	a	idempotence of +
a.(b+c)	=	a.b + a.c	left distribution of . over +
(a+b).c	=	a.c + b.c	right distribution of . over +
1.a	=	a	1 is a left unit for .
a.1	=	a	1 is a right unit for .
a*	=	1 + a*.a	
a*	=	(1 + a)*	

and a = b*.c is a solution of a = b.a + c.

A regular algebra with a zero, 0, also satisfies

0.a = 0	0 is a left zero for .
a.0 = 0	0 is a right zero for .
0+a = a	0 is a unit for +

It is also true that

$$a^* = \sum_{i=0}^{i=\infty} a.a..... \quad (i \text{ factors})$$

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Memorandum 3748	3. Agency Reference	4. Report Security Classification UNCLASSIFIED
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location Royal Signals and Radar Establishment		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title

CHECKING MICROCODE ALGEBRAICALLY

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
Foster J M				

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This Memorandum describes algebraic methods used by a program which processes microcode based on the AMD 2910 in order to find some of its properties. The methods could be applied to other controllers. Examples of the properties which can be found are given, including checking for timing constraints, ensuring that interrupts are polled frequently, checking against expression stack overflow and ensuring the absence of certain sequences of instruction. The method separates into a part which deals only with the control structure, for this program that of the AMD 2910, and a part which deals with the operations performed by the microinstructions, in this case those of the ICL Perq. It has been used to check many properties of the implementation of Flex on Perq, which involves more than 4000 microinstructions.

SR 1748

END

FILMED

2-85

DTIC